Facharbeit FS 2014

The Multigranular Temporal Nearest Neighbor Join Operator

Universität Zürich
Departement of Informatics
Prof. Dr. Michael Böhlen

von
Erik Hasselberg
Zelgstrasse 11, 8134 Adliswil
Schweiz

July 30, 2014

Contents

1 Introduction

In this report we will investigate the temporal nearest neighbor join (TNNJ) with multiple time granularities. Given an outer relation **r** of schema $R = [E, G, T]$ and an inner relation **s** of schema $S = [E, G, T, M]$ the TNNJ operator computes an equijoin on E and G, and a nearest neighbor join on T with group G for the tuples of the outer relation without equijoin. A timestamp T is an interval with a starting point T_S and an ending point T_E. Both, T_S and T_E and the granularity $gran$ itself are stored in a tuple of **r** and **s**. So we say that $T = [label, T_S, T_E, gran]$, for example $[Summer2013, 20130621, 20130921, 2]$.

r

T			
$label$	T_S	T_E	$gran$
20120705	20120705	20120705	0
June 2013	20130601	20130630	1
20140228	20140228	20140228	0
August 2014	20140801	20140831	1

s

T			
$label$	T_S	T_E	$gran$
2011	20110101	20111231	3
July 2011	20110701	20110731	1
20120721	20120721	20120721	0
2013	20130101	20131231	3
20130207	20130207	20130207	0
April 2013	20130401	20130430	1
Summer 2013	20130621	20130921	2
20131230	20131230	20131230	0
Spring 2014	20140320	20140620	2
April 2014	20140401	20140430	1
20140429	20140429	20140429	0
October 2014	20141001	20141031	1
2015	20150101	20151231	3

Figure 1: Input relations **r** and **s** sorted by T_S

The goal of this report is to design, define and implement an extension to the TNNJ operator that deals with relations having a muligranular timestamp. This includes to define a function name that calculates the distance between a **r** and a **s** tuple. Finally a new algorithm will be implemented, using the defined distance function, in the kernel of PostgreSQL to find the nearest neighbors for **r** in **s**.

An example data set has already been introduced in fig.1. Afterwards in this report we introduce the granularity. Then the multigranular distance function will be introduced and also two examples are given to explain the calculation. After checking the properties of the introduced multigranular distance function we propose a new order of the tables. Afterwards the procedure of the algorithm will be explained before discussing the algorithm itself in pseudocode. In the last section some experiments will be run to test and compare the runtime of the TNNJ algorithm.

2 Granularities

r and **s** are the two given input relations. Both store tuples, which consist as described above of a muligranular timestamp $T = [label, T_S, T_E, gran]$. We define the granualarity $gran$ as an integer such that:

$$\forall T_1, T_2 \in \triangle(t)(T_2.gran < T_1.gran) \rightarrow (T_2.T_E - T_2.T_S < T_1.T_E - T_1.T_S)$$

where $\triangle(t)$ it the time domain. If there are two timestamps T_1 and T_2 so that the granularity of T_2 is smaller than the granularity of T_1, the length of the interval of T_2 is also smaller than the length of the interval of T_1.

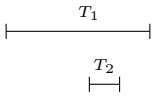


Figure 2: Definition of $gran$

In the tables of fig.1 we can see some examples of different timestamps. A timestamp can be a year, a season, a month or a day. A graphical representation of timestamps with different granularities is shown below.

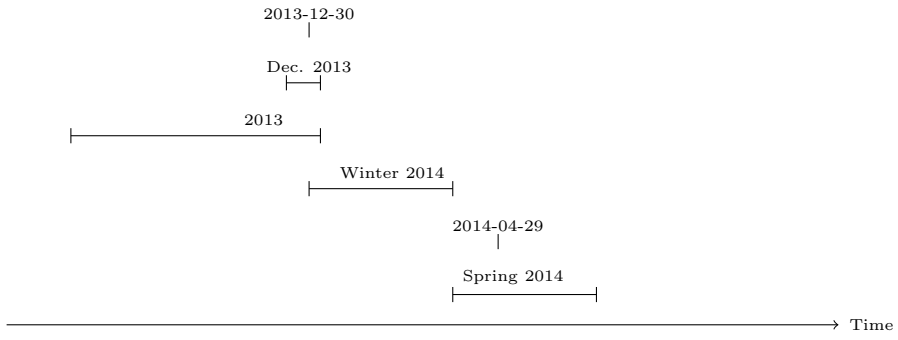


Figure 3: Example Granularities: year, season, month, day

To these timestamps now the granularity is associated so they can be differentiated. The year ($365 \leq$ length ≤ 366) gets $gran = 3$, the season ($87 \leq$ length ≤ 95) $gran = 2$, month ($28 \leq$ length ≤ 31) granularity is $gran = 1$ and the day (length=0) is $gran = 0$. The timpestamp with the longest interval, i.e. the largest

granulartiy, preserves the biggest granularity value (which is 3), while the timestamp with the shortest interval, i.e. the finest granularity, obtains the smallest granularity value (which is 0). Since the euclidean distance cannot be applied to intervals, how do we calculate now the distance between the tuples in **r** and **s**?

3 Time management

In this section we give a definition of the distance function and explain how we calculate the distance in different cases. Also we give two examples for a better understanding of the calculation.

3.1 Defintion of the med function

As we have seen before a table stores timestamps of different granularities. For computing the distance between two tuples we introduce the function *med(r,s,p)*, i.e. Multigranular Euclidean Distance, that returns the distance from r to s where a value $0 \leq p \leq 1$ specifies which distance should be considered. For example as you can see in fig.4, for $p = 0$ we take the shortest possible distance between r and s, for $p = 1$ we calculate the longest distance between r and s; for $p = 0.5$ we look at the intermediate distance, i.e. the distance between the shortest and the longest; etc. In the three examples in the figure we can see that if $p < 0.5$ we are looking at a case which is closer to the shortest possible distance and if $p > 0.5$ we consider a case closer to the furthermost distance between r and s.
Since the timestamp of a tuple is an interval, r and s can overlap each other. We first need to define what overlapping means so we can afterwards distinguish better between more possible cases in calculating the distance between r and s.

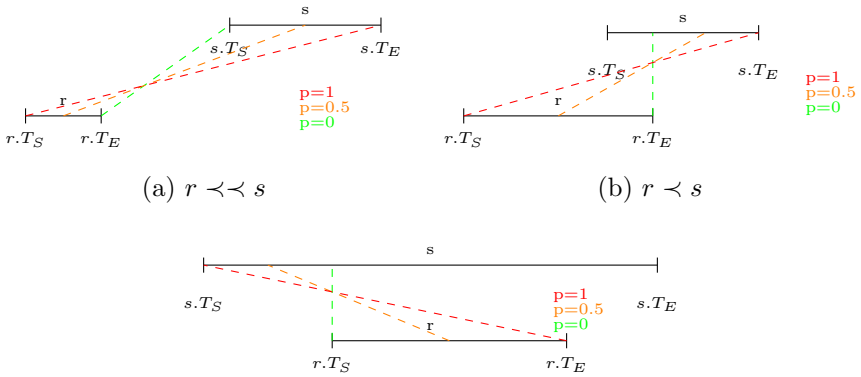
3.1.1 Definition of overlapping

$$r \prec\prec s \Leftrightarrow r.T_E < s.T_S, \qquad \text{left disjoint} \qquad (1)$$

$$r \prec s \Leftrightarrow r.T_S < s.T_S \wedge s.T_S < r.T_E \wedge r.T_E < s.T_E \qquad \text{left overlapping} \qquad (2)$$

$$r \subset s \Leftrightarrow s.T_S < r.T_S \wedge s.T_E > r.T_E \qquad \text{left containing} \qquad (3)$$

In the definition there are three different cases from which we can distinguish shown in fig.4. Right disjoint ($r \succ\!\succ s$), right overlapping ($r \succ s$) and right contained ($r \supset s$) are defined symmetrically.



(a) $r \prec\!\prec s$ (b) $r \prec s$

(c) $r \subset s$

Figure 4: Different examples of r and s regarding overlapping

3.1.2 med function

$$med(r,s,p) = \begin{cases} d\Big(r.T_E - p(r.T_E - r.T_S), s.T_S + p(s.T_E - s.T_S)\Big) & r \prec\!\prec s \\ d\Big(r.T_S + p(r.T_E - r.T_S), s.T_E - p(s.T_E - s.T_S)\Big) & r \succ\!\succ s \\ d\Big(0, p(s.T_E - r.T_S)\Big) & r \prec s \\ d\Big(0, -p(r.T_E - s.T_S)\Big) & r \succ s \\ max\Big(d\Big(0, p(s.T_E - r.T_S)\Big), d\Big(0, -p(r.T_E - s.T_S)\Big)\Big) & else \end{cases}$$

The *med* functions uses the euclidean distance function d to calculate the distance between two time points. d is defined as the absolute value of the numerical difference of two points on the real line. Thus if x and y are two points on the real line, the distance d between them is given by:

$$\sqrt{(x-y)^2} = |x - y|$$

Finally it has to be said that the *med* function returns the distance always in the smallest granularity.

The formulas for calculating the *med* when r and s are disjoined derive from the figure 4a. For the case when $r \prec s$ the formula we give is the short form of $d\Big(0, p(s.T_E - r.T_S)\Big) =$

$d\Big(r.T_E - p(r.T_E - r.T_S), r.T_E - p(r.T_E - r.T_S) + p(s.T_E - r.T_S)\Big)$. The given formula for $r \succ s$ has the same short form. The "else" case represents the situations when $r \subset s$ or $r \supset s$.

3.2 Examples

The examples below show for two cases, $r \prec\prec s$ and $r \prec s$, the results of the *med* function for three values of p (0, 0.5, 1), one step-by-step calculation and a graphic of the situation.

Disjoint

1. $med(August2014, October2014, p = 0) = 31\ days$
2. $med(August2014, October2014, p = 0.5) = 61\ days$
3. $med(August2014, October2014, p = 1) = 91\ days$

$$med(August2014, October2014, 0.5) =$$
$$d(2014.08.31 - 15days, 2014.10.01 + 15days) =$$
$$d(2014.08.16, 2014.10.16) = 61\ days$$

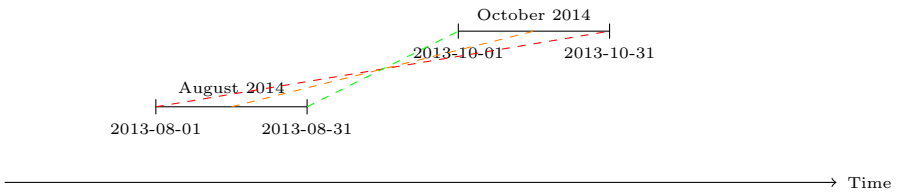


Figure 5: Example $r \prec\prec s$ with real dates

As it can be seen in figure 5 the case above describes the case $r \prec\prec s$ because August 2014 starts and ends before October 2014. So the first formula is applied to calculate the distance. We see that in the case for $p = 0$ the shortest possible distance is between the ending point of August 2014 and the starting point of October 2014 which corresponds to the length of a month (September 2014).

Overlapping

1. $med(June2013, Summer2013, p = 0) = 0\ days$
2. $med(June2013, Summer2013, p = 0.5) = 56\ days$
3. $med(June2013, Summer2013, p = 1) = 112\ days$

$$
\begin{aligned}
med(June2013, Summer2013, 1) = \\
d(2013.06.01, 2013.06.01 + 1 * (2013.09.21 - 2013.06.01)) = \\
d(2013.06.01, 2013.06.01 + 112days) = \\
d(2013.06.01, 2013.09.21) = 112\ days
\end{aligned}
$$

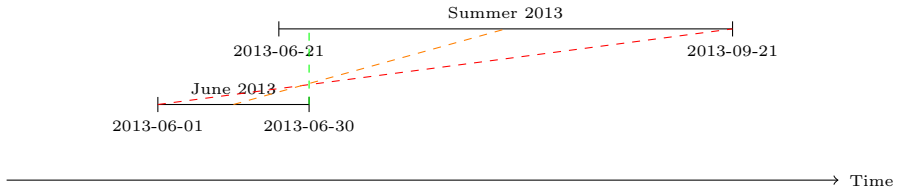


Figure 6: Example $r \prec s$ with real dates

In this example where $r \prec s$, the shortest distance in the case for $p = 0$ is 0 days. This is because June 2013 is a summer month.

3.3 Properties of the med function

Since the sort merge NNJ works with metric, we check if *med* is a metric. Per definition a metric on a set $\mathbf{X}$ is a function $d : \mathbf{X} \times \mathbf{X} \to \mathbb{R}$ (where $\mathbb{R}$ is the set of real numbers). For all x, y, z in $\mathbf{X}$, this function is required to satisfy the following conditions:

1. $d(x, y) \geq 0$ (non-negativity)
2. $d(x, y) = 0$ if and only if $x = y$ (identity)
3. $d(x, y) = d(y, x)$ (symmetry)
4. $d(x, z) \leq d(x, y) + d(y, z)$ (triangle inequality)

Hereafter every single condition needs to be checked if it can hold for *med*.

First condition

The distance between two tuples can never be smaller than zero. If you look at example 2, where r starts after s and so the distance on the time axis could be negative, the d function takes the absolute value.

1. $med(2013, 2015, p = 0) = 366$ days
2. $med(2016, 2014, p = 1) = 1095$ days

Second condition

For the second condition, which says that the distance is only equal to zero when the two tuples are equivalent, we give two counter examples to show, that this condition doesn't hold for med. In the first example, where $r = s$ we see that the distance is not zero when $p = 1$. As we see in the second example for $p = 0$ and $r \neq s$ the distance instead is zero.

1. $med(2013, 2013, p = 1) = 364$ days
2. $med(20130101, 2013, p = 0) = 0$ days

Third condition

The symmetry condition for the med function holds. The proof is given afterwards. First we look at the case when $r_1 \prec\prec s_1$ and see that $med(r_1, s_1, p) = med(s_1, r_1, p)$.

1. $med(r_1, s_2, p) \stackrel{?}{=} med(s_1, r_1, p)$

$$d\Big(r_1.T_E - p(r_1.T_E - r_1.T_S), s_1.T_S + p(s_1.T_E - s_1.T_S)\Big) \stackrel{?}{=}$$
$$d\Big(s_1.T_E - p(s_1.T_E - s_1.T_S), r_1.T_S + p(r_1.T_E - r_1.T_S)\Big) \Rightarrow$$
$$d\Big(r.T_S + p(r.T_E - r.T_S), s.T_E - p(s.T_E - s.T_S)\Big)$$

Thereafter we look at the case when $r_2 \prec s_2$ and also note here that $med(r_2, s_2, p) = med(s_2, r_2, p)$.

2. $med(r_2, s_2, p) \stackrel{?}{=} med(s_2, r_2, p)$

$$d\Big(0, p(s_2.T_E - r_2.T_S)\Big) \stackrel{?}{=} d\Big(0, p(r_2.T_E - s_2.T_S)\Big) \Rightarrow$$
$$d\Big(0, -p(r.T_E - s.T_S)\Big)$$

Completing we look at the case when $r_3 \subset s_3$ and can observe also here that $med(r_3, s_3, p) = med(s_3, r_3, p)$.

3. $med(r_3, s_3, p) \stackrel{?}{=} med(s_3, r_3, p)$

$$max\Big(d\Big(0, p(s_3.T_E - r_3.T_S)\Big), d\Big(0, -p(r_3.T_E - s_3.T_S)\Big)\Big) \stackrel{?}{=}$$
$$max\Big(d\Big(0, p(r_3.T_E - s_3.T_S)\Big), d\Big(0, -p(s_3.T_E - r_3.T_S)\Big)\Big) \Rightarrow$$
$$max\Big(d\Big(0, p(s_3.T_E - r_3.T_S)\Big), d\Big(0, -p(r_3.T_E - s_3.T_S)\Big)\Big)$$

For the cases where $r_1 \succ\succ s_1, r_2 \succ s_2, r_3 \supset s_3$ the procedure is analogus. So we can say, that the symmetry condition holds for the *med* function.

Fourth condition

For the fourth condition the counter example given below shows, that *med* doesn't hold the triangle inequality.

1. $med(2013, 2015, p = 1) \stackrel{?}{=}$
   $med(2013, 2014, p = 1) + med(2014, 2015, p = 1) \Rightarrow$
   1094 days $\neq$ 729 days+729 days

In the example for $p = 1$ we consider the greatest possible distance between the two tuples. So we count the length of the interval of the tuple 2014 twice because primarily we calculate the distance between 2013 and 2014, what means for $p = 1$ we consider the dates 2013-01-01 and 2014-12-31. For the second subtotal we calculate the distance from 2014-01-01 to 2015-12-31. So in total the sum of the two distances is four years and not three as it had to be to fulfill the fourth condition of a metric.

3.4 Implications

With multigranular timestamps there doesn't exist a total order on T because we cannot sort the tuples so that all nearest neighbors (NN) are one after the other. To demonstrate this we introduce two example relations **r** and **s**, shown in fig.7.

r

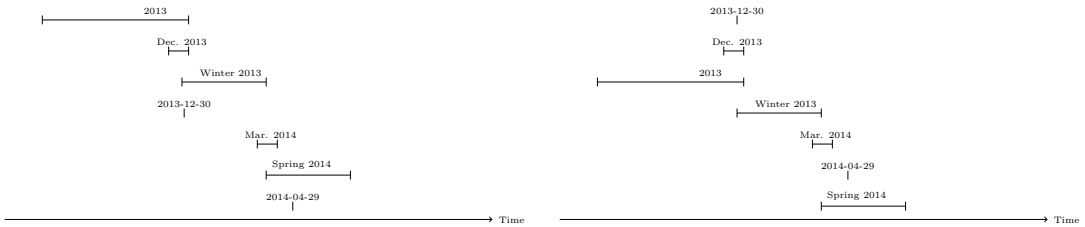
	T			
	label	T_S	T_E	*gran*
r_1	20140228	20140228	20140228	0
r_2	2014	20140101	20141231	3

s

	T			
	label	T_S	T_E	*gran*
s_1	20131230	20131230	20131230	0
s_2	Winter 2013	20131221	20140319	2
s_3	December 2013	20131201	20131231	1
s_4	2014	20140101	20141231	3
s_5	March 2014	20140301	20140331	1
s_6	Spring 2014	20140320	20140620	2
s_7	20140429	20140429	20140429	0

Figure 7: **r** and **s**

The tuples overlap each other: In fig.8 below we illustrate that when we sort $\mathbf{s}$ by T_S or by T_E the order of the tuples is a total different one.



(a) ordered by T_S (b) ordered by T_E

Figure 8: No total order exists

If you consider r_1, after fetching its first nearest neighbor s_1 we need to scan a lot of tuples before fetching the second nearest neighbor, i.e. s_7. Then we need to go back at the beginning of $\mathbf{s}$ for fetching the nearest neighbors of r_2 $(s_2, s_4, s_5, s_6, s_7)$: We want to avoid to refetch tuples that are not nearest neighbors and we want the nearest neighbors being always one after the other.

4 The Algorithm

4.1 Introduction

Our idea is to sort the input tables also by the granularity *gran* and not just T_S. In such a case we achieve that there can't be any additional tuples between two nearest neighbors of the same granularity in $\mathbf{s}$. We first define $\mathbf{r}_i$ and $\mathbf{s}_j$ sucht that:

$$\mathbf{r}_i = \sigma_{gran=i}(\mathbf{r})$$
$$\mathbf{s}_j = \sigma_{gran=j}(\mathbf{s})$$

r

	T			
	label	T_S	T_E	*gran*
$\mathbf{r}_0$	20120705	20120705	20120705	0
	20140228	20140228	20140228	0
$\mathbf{r}_1$	June 2013	20130601	20130630	1
	August 2014	20140801	20140831	1
$\mathbf{r}_3$	2012	20120101	20121231	3
	2014	20140101	20141231	3

s

	T			
	label	T_S	T_E	*gran*
$\mathbf{s}_0$	20120721	20120721	20120721	0
	20131230	20131230	20131230	0
	20140429	20140429	20140429	0
$\mathbf{s}_1$	July 2011	20110701	20110731	1
	April 2013	20130401	20130430	1
	April 2014	20140401	20140430	1
$\mathbf{s}_2$	Spring 2014	20140320	20140620	2
$\mathbf{s}_3$	2011	20110101	20111231	3
	2013	20130101	20131231	3
	2015	20150101	20151231	3

Figure 9: Input relations $\mathbf{r}$ and $\mathbf{s}$ sorted by $granularity, T_S$: the sorting on *granulartiy* allows to build the partitions $\mathbf{r}_i$ and $\mathbf{s}_j$

For example now, for the tuple 2014 in $\mathbf{r}_0$ no other tuple can be between its nearest neighbors in $\mathbf{s}_0$, i.e. 2013, 2015. The following figure is nearly the same as fig.8, with the difference that the tuples are now sorted by the granularity *gran* and T_S. So we achieve that there is now a total order between all tuples of the same granularity, i.e. tuples of different years will come one after the other, tuples of different seasons will come consecutively etc.

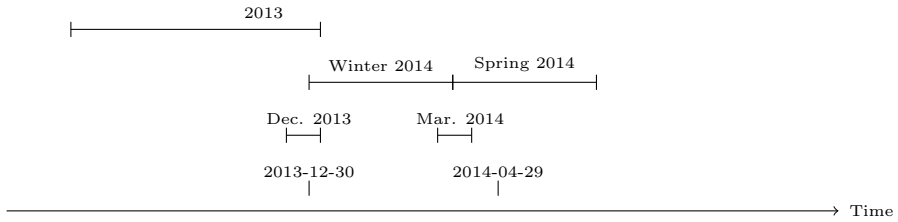


Figure 10: Total order between tuples of same granularity

Since in every partition $\mathbf{s}_j$ there now exists a total order on T, the nearest neighbors of every $\mathbf{r}$ tuple are always one after the other. Therefore we join each tuple in $\mathbf{r}_i$ with its *local* nearest neighbors, i.e. the nearest neighbors in every $\mathbf{s}_j$. At the end, once all local nearest neighbors have been found, we select as nearest neighbors only the tuples with the smallest distance.

Algorithm 1: TNNJ

```
1 begin
2     foreach r_i ∈ r do
3         foreach s_j ∈ s do
4             z ← z ∪ Π_{R,S,med(R,S,p)}(r_i ⋈^{NN(T)}_{EQ(G)} s_j)
5     return (_R ϑ_{MIN(d)})(z) ⋈ z
```

4.2 Procedure

We see looking at line 4 of the algorithm 1 that we can fetch $\mathbf{r}_i$ and $\mathbf{s}_j$ using two different ways.

The first uses an index on the granularity *gran* to fetch $\mathbf{r}_i$ and $\mathbf{s}_j$ without a scan of the relation (monogranular version). This approach has two disadvantages:

1. Blocks storing tuples of different granularitites have to be fetched multiple times
2. An index scan on a complete relation (all tuples are needed) is always slower than a sequential scan

The second approach is to implement an efficient procedure using sort merge (multigranular version), that fetches all tuples once at the beginning. This implementation will be described herafter:
We continuously scan $\mathbf{r}_0$ to fetch its nearest neighbors in $\mathbf{s}_0$. When we reach a new partition in $\mathbf{r}$ (i.e. $\mathbf{r}_1$), we do a restore in $\mathbf{r}$ (go to first tuple of $\mathbf{r}_0$) and scan $\mathbf{s}_0$ until we reach $\mathbf{s}_1$. We then repeat the process. As soon as we have joined $\mathbf{r}_0$ with all $\mathbf{s}$ partitions, we go to the first tuple of $\mathbf{r}_1$ and rescan $\mathbf{s}$.
Since the various partitions $\mathbf{r}_i, \mathbf{s}_j$ are not stored in different tables, we can't just run line 4 of Algorithm 1, i.e at every step we would need to fully scan $\mathbf{r}$ and $\mathbf{s}$ for fetching $\mathbf{r}_i$ and $\mathbf{s}_j$. Since $\mathbf{r}$ and $\mathbf{s}$ are sorted by *gran*, we always know when $\mathbf{r}_i$ and $\mathbf{s}_j$ start and finish. Now we present the join result between relations $\mathbf{r}$ and $\mathbf{s}$ of fig.9 with $p = 0$. The following table in fig.11 shows all local nearest neighbors for every $\mathbf{r}_i$ in every $\mathbf{s}_j$.

z

	r				**s**				
	label	T_S	T_E	*gran*	*label*	T_S	T_E	*gran*	*d*
	20120705	20120705	20120705	0	20120721	20120721	20120721	0	16
	20140228	20140228	20140228	0	20131230	20131230	20131230	0	60
	20140228	20140228	20140228	0	20140429	20140429	20140429	0	60
	20120705	20120705	20120705	0	April 2013	20130401	20130430	1	270
$\mathbf{r}_0$	20140228	20140228	20140228	0	April 2014	20140401	20140430	1	32
	20120705	20120705	20120705	0	Spring 2014	20140320	20140620	2	632
	20140228	20140228	20140228	0	Spring 2014	20140320	20140620	2	20
	20120705	20120705	20120705	0	2013	20130101	20131231	3	180
	20140228	20140228	20140228	0	2013	20130101	20131231	3	180
	June 2013	20130601	20130630	1	20131230	20131230	20131230	0	183
	August 2014	20140801	20140831	1	20140429	20140429	20140429	0	94
	June 2013	20130601	20130630	1	April 2013	20130401	20130430	1	32
$\mathbf{r}_1$	August 2014	20140801	20140831	1	April 2014	20140401	20140430	1	93
	June 2013	20130601	20130630	1	Spring 2014	20140320	20140620	2	263
	August 2014	20140801	20140831	1	Spring 2014	20140320	20140620	2	42
	June 2013	20130601	20130630	1	2013	20130101	20131231	3	0
	August 2014	20140801	20140831	1	2015	20150101	20151231	3	123
	2012	20120101	20121231	3	20120721	20120721	20120721	0	0
	2014	20140101	20141231	3	20140429	20140429	20140429	0	0
	2012	20120101	20121231	3	April 2013	20130401	20130430	1	91
	2014	20140101	20141231	3	April 2014	20140401	20140430	1	0
$\mathbf{r}_3$	2012	20120101	20121231	3	Spring 2014	20140320	20140620	2	444
	2014	20140101	20141231	3	Spring 2014	20140320	20140620	2	0
	2012	20120101	20121231	3	2011	20110101	20111231	3	1
	2012	20120101	20121231	3	2013	20130101	20131231	3	1
	2014	20140101	20141231	3	2013	20130101	20131231	3	1
	2014	20140101	20141231	3	2015	20150101	20151231	3	1

Figure 11: Output table with all local nearest neighbors: $\bigcup_{i,j} \mathbf{r}_i \overset{\text{NN(T)}}{\underset{\text{EQ}(\varnothing)}{\bowtie_{\varnothing}}} \mathbf{s}_j$

After founding all local nearest neighbors for every $\mathbf{r}_i$, we can rescan the result in order to select the closest local nearest neighbors. The table in fig.12 shows the final output result.

y

r				**s**				
label	T_S	T_E	*gran*	*label*	T_S	T_E	*gran*	*d*
20120705	20120705	20120705	0	20120721	20120721	20120721	0	16
20140228	20140228	20140228	0	Spring 2014	20140320	20140620	2	20
June 2013	20130601	20130630	1	2013	20130101	20131231	3	0
August 2014	20140801	20140831	1	Spring 2014	20140320	20140620	2	42
2012	20120101	20121231	3	20120721	20120721	20120721	0	0
2014	20140101	20141231	3	April 2014	20140401	20140430	1	0
2014	20140101	20141231	3	20140429	20140429	20140429	0	0
2014	20140101	20141231	3	Spring 2014	20140320	20140620	2	0

Figure 12: Final output table with the closest local nearest neighbors: $\mathbf{r} \bowtie \mathbf{s}$

The total runtime of the algorithm therefore can be expressed as:

$$G * (\mathbf{r} + \mathbf{s})$$

No additional costs for calculating $T_S, T_E, gran$ starting from its label have to be added because this "exctractions" process can be done offline. For example to set the granularity of all timestamps with a day or year label we can run the following PostgreSQL commands:

```
UPDATE S SET GRAN=0 WHERE LENGTH(LABEL)=8; // dd granularity

UPDATE S SET GRAN=3 WHERE LENGTH(LABEL)=4; // yy granularity
```

4.3 State Machine

The algorithm is implemented extending the routine (i.e. a set of states) for traditional Sort MergeJoin; the algorithm is composed by 7 main states, each of those will be described in the following subsections.

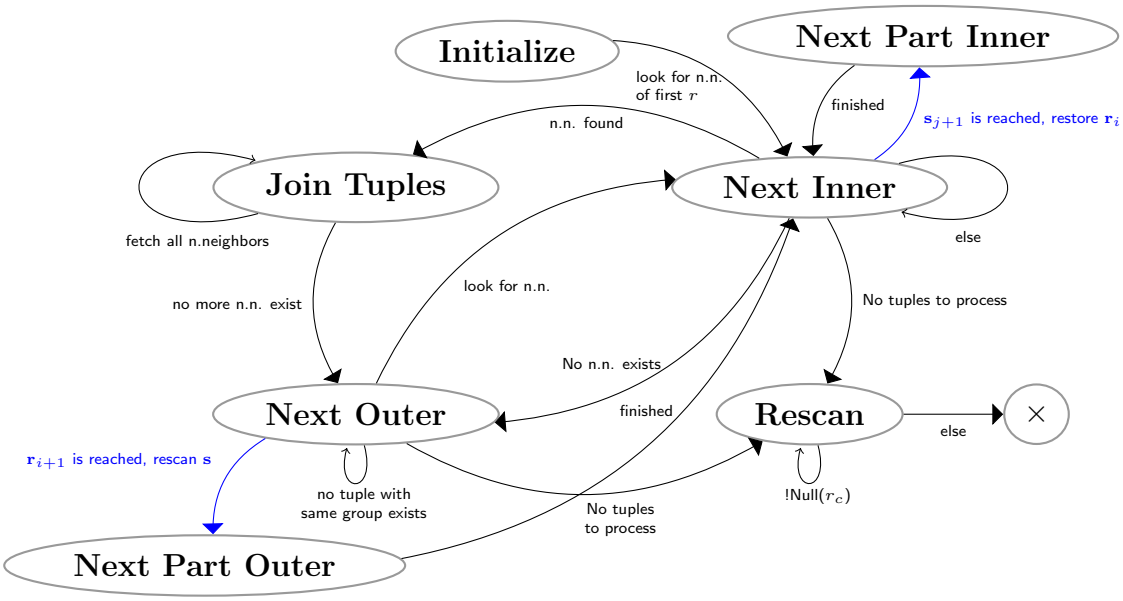


Figure 13: State Diagram: TNNJ

We start with the Initialize and look then for the nearest neighbors of the first tuple in $\mathbf{r}_0$ in NextInner. We stay so long in NextInner until we have found a nearest neighbor and go to JoinTuples. After joining the $\mathbf{r}_0$ tuple with its nearest neighbor(s) from $\mathbf{s}_0$ we fetch the next tuple in NextOuter. For this tuple we look again for its nearest neigbor(s) as described above. When we reach the next partition in $\mathbf{s}$ (i.e. $\mathbf{s}_1$), we go to NextPartitionInner where we reinitialize $\mathbf{r}$ with the first tuple of the current partition

(i.e. $\mathbf{r}_0$) and initialize $\mathbf{s}$ with the first tuple of the next partition $\mathbf{s}_1$. Again, in NextInner we look for the neareast neigbors for the $\mathbf{r}_0$ tuples in $\mathbf{s}_1$. This procedure is repeated for every partition of $\mathbf{s}$.
Once all $\mathbf{s}$ partitions have been processed for $\mathbf{r}_0$, we go in NextPartitionOuter. In this state we reach the next partition of $\mathbf{r}$ (i.e. $\mathbf{r}_1$) and rescan $\mathbf{s}$ (fetching again $\mathbf{s}_0$). Like this we're running the algorithm until we reach the end of $\mathbf{r}$ and $\mathbf{s}$, i.e. the last partitions and there are no tuples left to process. We then rescan the result to select only the closest nearest neighbors.

4.4 Initialize

This state is the first to be executed. We sort the outer relation $\mathbf{r}$ and the inner relation $\mathbf{s}$ by $(gran, K1, K3, K2)$[1]. We start the scan of the two relations initializing r with the first $\mathbf{r}$ tuple, and s_c, s_n, and s_p with the first $\mathbf{s}$ tuple. The algorithm will set, during its iteration s_c to the current, s_n to the next and s_p to the previous $\mathbf{s}$ tuple. Needed flags are initialzied with 0. Before changing the state and going to NextInner, we mark the first position in $\mathbf{r}$ and $\mathbf{s}$ since we work with backtracking and need to restore the positions of both in some cases we'll describe later in detail.

State 1: Initialize

Input: $\mathbf{r}, \mathbf{s}, K1, K2, K3, gran$

```
1  begin
2      Sort(r);  Sort(s)
3      r ← fetchRow(r)  // Current r tuple
4      markPosition(r)
5      s_c ← fetchRow(s  // Current s tuple
6      markPosition(s)
7      s_n ← s_c  // Next s tuple
8      s_p ← s_c  // Previous s tuple
9      goToEndInner ← 0 // Set flags
10     holdOuter ← 0
11     go to NextInner
```

[1] $K2$ stands for the identifier E where $K3$ stands for the time T.

4.5 Next Inner

In this state we fetch the next **s** tuple. In lines 4-7 we check if an inner tuple with the same group of the actual outer tuple exists at all: if not, no join match for it exists and we go directly to the state NextOuter to fetch a new row in **r**.
In lines 8-16 we fetch a new inner tuple. Before actually fetching a new row we check if r and s_n belong to the same group, if s_c and s_n have the same granularity and if r is closer to s_n than to s_c, then s_n *may* be its first nearest neighbor. Therefore we mark its position in line 11, before the previous and the current tuple are updated. We mark the position too when a new partition $\mathbf{s}_{j+1}$ is reached (line 15). Then we check if we have reached a new partition in **s** (line 19): If so, we go to the State NextPartitionInner. When the init is done and we come back again to NextInner, we fetch a new row.
In lines 24-27 we check if r and s_c are an equijoin on $K1$ and $K2$. After returning the equijoin match we check if there doesn't exist a further equijoin and if this is the case we change the state and go to NextOuter.
In lines 28-32, as soon as the distance grows, we reached the end of **s**, a different group tuple or a tuple with a new granularity is fetched, we are sure that the previously marked tuple is its first nearest neighbor (the tuple marked in line 11), therefore we restore its position and we go to the state JoinTuples.
The next if statement we only check when a new partition $\mathbf{r}_{i+1}$ (there the *goToEndInner* flag has been set) in **r** has been reached and we've reached the end of **s**.[2] We change the value of *goToEndInner* to set the second flag *holdOuter* which we need to ensure not fetching a next tuple in $\mathbf{r}_{i+1}$. If r_n and s_n are both not null we go again to NextInner. In line 18 we now set *holdOuter*, in line 22 we reset *goToEndInner* and after staying once more in NextInner we enter the first if statement and go afterwards to the state NextOuter.
As is has been said before, when there are no tuples in **r** and **s** left to proceed, we change the state and go to Rescan to fetch only the nearest neighbors with the smallest temporal distance from the output **z**.

[2] Compare state NextOuter, lines 11-13, for more details.

State 2: NextInner

```
1  begin
2      if holdOuter = 1 ∧ goToEndInner = 0 then
3          go to NextOuter
4      if !Null(r) ∧ r.K1 < s_c.K1 ∧ goToEndInner = 0 then
5          go to NextOuter  // No NN exists
6      if !Null(r) ∧ Null(s_n) ∧ r.K1 > s_c.K1 ∧ goToEndInner = 0 then
7          go to NextOuter  // No NN exists
8      if !Null(s_n) then
9          if !Null(r) ∧ r.K1 = s_n.K1 ∧ s_c.gran = s_n.gran ∧ goToEndInner = 0 ∧ (
10             d(r, s_c) > d(r, s_n) ∨ r.K1 ≠ s_c.K1) then
11             markPosition(s)  // Mark first NN
12         s_p ← s_c
13         s_c ← s_n
14         if s_c.gran > s_p.gran then
15             markPosition(s)  // Mark first tuple of new partition s_{j+1}
16         s_n ← fetchRow(s)
17     if goToEndInner = −1 then
18         holdOuter ← 1  // Last tuple in r_i has been reached
19     if s_c.gran > s_p.gran ∨ goToEndInner = −1 then
20         if holdOuter = 0 then
21             go to NextPartitionInner  // restore r, initialize s_{j+1}
22         goToEndInner ← 0
23         go to NextInner
24     if !Null(r) ∧ r.K1 = s_c.K1 ∧ r.K2 = s_c.K2 ∧ goToEndInner = 0 then
25         z ← z ∪ (r ∘ s_c)  // Equijoin match
26         if Null(s_n) ∨ !(r.K1 = s_n.K1 ∧ r.K2 = s_n.K2) then
27             go to NextOuter
28     if !Null(r) ∧ r.K1 = s_c.K1 ∧ goToEndInner = 0 ∧ (
29         Null(s_n) ∨ d(r, s_c) < d(r, s_n) ∨ r.K1 ≠ s_n.K1 ∨ s_c.gran ≠ s_n.gran) then
30         s_c ← restorePosition(s)  // Fetch first NN
31         s_n ← fetchRow(s)
32         go to JoinTuples
33     if Null(s_n) ∧ goToEndInner = 1 then
34         goToEndInner ← −1  // When in the last partition of s
35     if Null(r) ∧ Null(s_n) then
36         Sort(z) by r.gran, r.K1, r.K3, r.K2, d
37         go to Rescan  // Both relations are empty, select closest NN
38     go to NextInner
```

4.6 Next Partition Inner

In this state a new $\mathbf{s}_j$ needs to be initialized. When a new partition $\mathbf{s}_{j+1}$ has been reached we restore $\mathbf{r}$ to the first tuple of $\mathbf{r}_i$ and initialize $\mathbf{s}$ to the first tuple of the next partition $\mathbf{s}_{j+1}$. s_p, s_n are both set to this tuple. Afterwards we reset the *goToEndInner* flag and go back again to NextInner.

State 3: NextPartitionInner

```
1 begin
2     r ← restorePosition(r)  // first r_i tuple
3     s_c ← restorePosition(s)  // first s_{j+1} tuple
4     s_p ← s_c
5     s_n ← s_c
6     goToEndInner ← 0
7     go to NextInner
```

4.7 Join Tuples

In this state we join r with all its nearest neighbors. We do this in line 2 and store r, its neareast neighbor s_c and the distance d between them in $\mathbf{z}$. Since the distance is a metric, as soon as the distance grows, the groups of the outer and the next inner tuple are different or we reach a new partition in $\mathbf{s}$ we are sure that no nearest neighbor for r exists anymore and we go to NextOuter. If there are still some nearest neighbors to fetch we remain in the state JoinTuples.

State 4: JoinTuples

```
1 begin
2     z ← z ∪ r ∘ s_c ∘ d(r, s_c)
3     if Null(s_n) ∨ d(r, s_n) > d(r, s_c) ∨ r.K1 ≠ s_n.K1 ∨ s_c.gran ≠ s_n.gran then
4         go to NextOuter  // No further NNs
5     else
6         s_c ← s_n
7         s_n ← fetchRow(s)
8         go to JoinTuples  // Still NNs to fetch
```

4.8 Next Outer

In this state, if the flag *holdOuter* is not set, we fetch a new **r** tuple r_c. If r_c corresponds to a new partition $\mathbf{r}_{i+1}$ and **s** has been scanned until the end, the join between $\mathbf{r}_i$ and all partitions of **s** has been completed. Therefore we go to the state NextPartitionOuter where we will reinitialize **r** with $\mathbf{r}_{i+1}$ and **s** with $\mathbf{r}_0$.
To ensure that when coming from the state NextInner, where the *holdOuter* flag[3] is set, we don't fetch a new **r** tuple at the beginning we reset the flag in line 9 (when coming the next time to NextOuter we are sure to fetch a new row at the beginning) and remain in the state NextOuter where we fetch then a new **r** tuple.
If we reached a new partition $\mathbf{r}_{i+1}$ in **r** but we haven't looked for the local neareast neighbors in all partitions of **s** for the tuples of $\mathbf{r}_i$, we need to scan **s** until we reach a new partition $\mathbf{s}_{j+1}$ and can restore **r** to the marked tuple (in line 6, NextPartitionOuter). So we set the flag *goToEndInner* and go to the state NextInner to fetch a new **s** tuple as long as a new partition $\mathbf{s}_{j+1}$ or the end of **s** has been reached.
We restore the **s** to its last marked position to fetch the nearest neigbor of r_c in the following cases:

1. r_c and r_n share the same nearest neighbors
2. we reached the end of **s** (last tuple of last partition
3. a new partition $\mathbf{r}_{i+1}$ has been reached

If we have reached the end of the last partition of **r** (i.e. r_c is null), but $\mathbf{s}_j$ isn't the last partition of **s** we go to NextInner for fetching $\mathbf{s}_{j+1}$. There as soon as $\mathbf{s}_{j+1}$ has been reached, we go to NextPartInner where we restore **r** to the first tuple of the last partition and **s** with $\mathbf{s}_{j+1}$. When we've reached the end of the last partition in both relations, we go to Rescan to select for each $r \in \mathbf{r}$ the closest among its local nearest neighbors.

[3]The *holdOuter* and also the *goToEndInner* flag are used when a new partition $\mathbf{r}_{i+1}$ has been reached but in **s** are still tuples to process. We want to be sure to not process $\mathbf{r}_{i+1}$ until all local nearest neighbors for $\mathbf{r}_i$ have been produced. So we do not fetch a new **r** tuple and we scan **s** until:

1. a new partition $\mathbf{s}_{j+1}$ is reached: we restore **r** to the first tuple of $\mathbf{r}_i$ and join $\mathbf{r}_i$ with $\mathbf{s}_{j+1}$
2. the end of **s** is reached: we rescan **s** and join $\mathbf{r}_{i+1}$ with $\mathbf{s}_0$

State 5: NextOuter

```
1  begin
2      if holdOuter ≠ 1 then
3          r_p ← r_c
4          r_c ← fetchRow(r)
5      if !Null(r_c) then
6          if Null(s_n) ∧ r_c.gran > r_p.gran then
7              go to NextPartitionOuter
8          if holdOuter = 1 then
9              holdOuter ← 0
10             go to NextOuter
11         if !Null(s_n) ∧ r_c.gran > r_p.gran then
12             goToEndInner ← 1 // Scan inner until new partition
13             go to NextInner
14         if (d(r_c, s_c) ≤ d(r_c, s_n) ∧ r_c.K1 = s_c.K1) ∨ Null(s_n) ∨ s_c.gran ≠ s_n.gran then
15             s_n ← restorePosition(s) // r_c has the same NN of r_p
16         go to NextInner
17     else if Null(r_c) ∧ !Null(s_n) then
18         go to NextInner  // still some tuples in s
19     else
20         Sort(z) by r.gran, r.K1, r.K3, r.K2, d
21         go to Rescan  // no tuples left to process, select closest NN
```

4.9 Next Partition Outer

This state will be executed when we have fetched all local nearest neighbors for $\mathbf{r}_i$ in every $\mathbf{s}_j$. In this state we rescan $\mathbf{s}$ what means we reinitialize $\mathbf{s}$ to $\mathbf{s}_0$ and set s_p, s_c, s_n to the first tuple of this partition again. We mark in both relations the position, in $\mathbf{r}$ the first tuple of the new partition $\mathbf{r}_{i+1}$, in $\mathbf{s}$ the first tuple in $\mathbf{s}_0$ again. Afterwards we go to the state NextInner.

State 6: NextPartitionOuter

```
1  begin
2      rescan(s)  // initialize s to the first partition s_0 again
3      s_c ← fetchRow(s) s_p ← s_c
4      s_n ← s_c
5      markPosition(s)
6      markPosition(r)  // first tuple of new partition r_{i+1}
7      go to NextInner
```

4.10 Rescan

This is the last state to be executed. We first fetch a new **z** tuple z_c[4]. In the Rescan we just print for r the closest among all its local nearest neighbors s_j. Since **z** is ordered by $(r.gran, r.K1, r.K3, r.K2, d)$ we are sure that the first local nearest neighbor of r is always the closest. So we return the current tuple as a join match every time it is not identical to the previous one (a new r is fetched) or if it has the same distance and is identical to the previous one (multiple nearest neighbors have occured for r). The algorithm ends in the state Rescan when no tuples to process are left.

State 7: Rescan

```
1 begin
2     z_c ← fetchRow(z)
3     if Null(z_c) then
4         End
5     if Null(z_p) ∨ z_p.K1 ≠ z_c.K1 ∨ z_p.K2 ≠ z_c.K2 ∨ z_p.K3 ≠ z_c.K3 ∨ z_p.d = z_c.d then
6         z_p ← z_c
7         y ← y ∪ z_c
8     go to Rescan
```

5 Experiments

In this section we test the implementation of our version, a multigranular, versus a monogranular algorithm and compare the runtime of both versions.

5.1 Random distribution

First we compare the runtime of the two algorithms on tables $|r|= 31000$ and $|s|= 31000$ where both tables contain tuples of all granularities ($|dd|= 10000$, $|mm|= 10000$, $|ss|= 10000$, $|yy|= 1000$). All tables are generated and also distributed randomly on the disk, as fig.14 illustrates below for example for **s**. A block can contain tuples of all granularities, just two, once or three.

[4]Remember that $z_c = r \circ s_j \circ d(r, s_j)$.

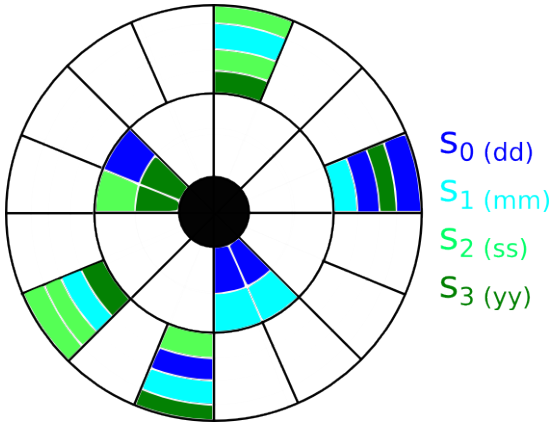


Figure 14: Random distribution of tuples on the disk: a block can contain tuples of all granularities, just two, once or three

In fig.15 the average run time of the multi- and the monogranular algorithm is shown. We can see, that the runtime for the mulitgranular algorithm with 3045ms is about 1.5 times faster than the monogranular version with a runtime of 5032ms.

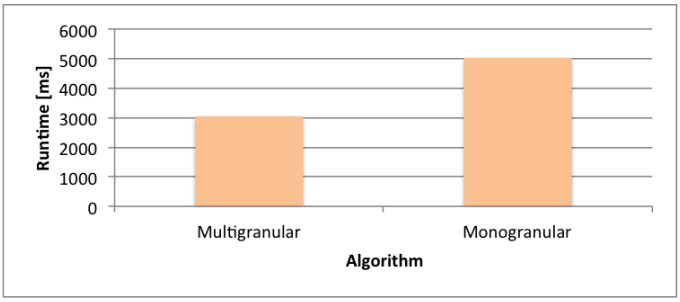


Figure 15: Runtime comparison for relation size=31000

In a second run, we increase the size of the tables **r** and **s** so that $|r|= 301000$ and $|s|= 301000$. So the size of all granularities except the year granularity is multiplied by ten, compared to the first run. The creation and distribution of the tables is also here done randomly.
In fig.16 we can see that the runtime of the multigranular algorithm is about 10 times faster than the monogranular version. The average runtime for the multigranular version is 110 seconds and for the monogranular version 1109 seconds.

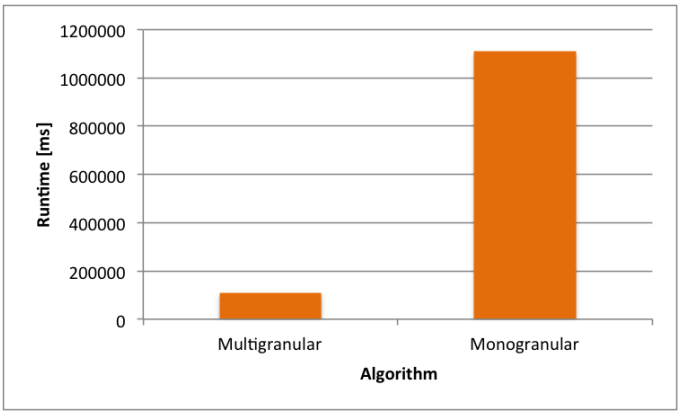


Figure 16: Runtime comparison for relation size=301000

The reason for the slow runtime of the monogranular algorithm can be found in the random distribution of the tuples. While the multigranular algorithm just fetchs all blocks once at the beginning, the monogranular version instead has to jump from block to block multiple times to fetch all tuples of different granularities.

5.2 Clustered distribution

In a second experiment we change the distribution of the tuples on the disk. The tuples are now clustered, which means that a block now only stores tuples of the same granularity. Fig.17 below shows this procedure. The size of the both relations **r** and **s** remains 301000.

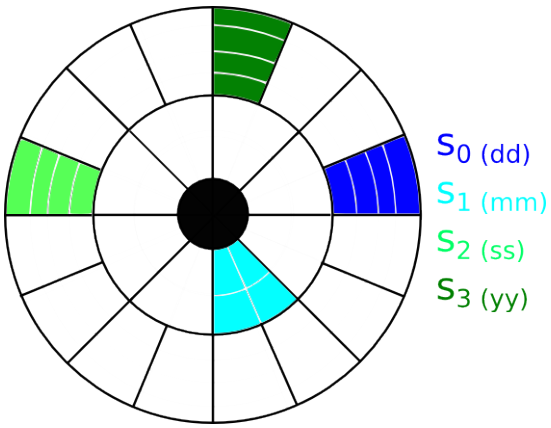


Figure 17: Clustered distribution of tuples on the disk: a block contains tuples of one granularity

In fig.18 we see that the runtime of the monogranular algorithm is about 7 times slower than the multigranular version. So the meassured runtime of the multigranular version is 111 seconds, of the monogranular version instead 740 seconds. The slowness of the monogranular algorithm can be explained based on the scan it uses. While the multigranular algorithm uses an sequential scan, which is faster when all tuples are needed, the monogranular version uses an index scan on the granularity. The costs for traversing the search three for every single tuple to look up which granularity it stores are so high, that a lot of calculation time on the CPU is needed: a sequential scan on a table is always faster than an index scan when all tuples are needed.

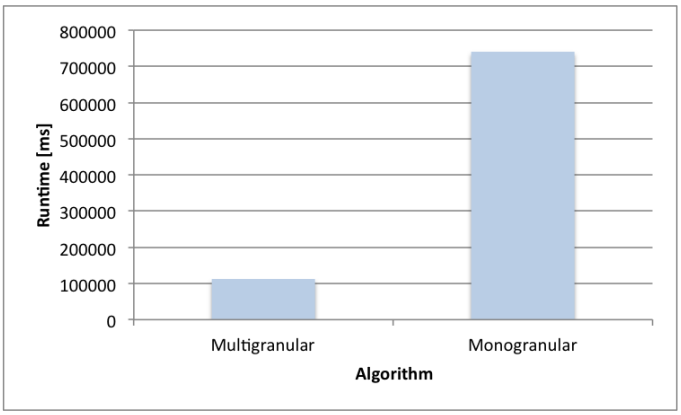


Figure 18: Runtime comparison clustered distribution

5.3 Totally unclustered distribution

Now we want to compare the runtime when the distribution of the tuples is totally unclustered, which means that every block now stores tuples of every granularity, as illustrated in fig.19. The size of both tables remains at 301000.

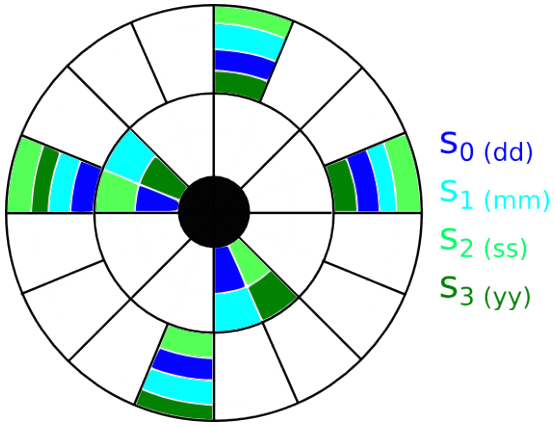


Figure 19: Totally unclustered distribution of tuples on the disk: a block contains tuples of all granularities

In fig.20 we see that the runtime for both versions is nearly exactly the same as in the clustered distribution. So here we can explain the slower runtime of the monogranular version again with the index scan it uses. The fact, that the runtime in both experiments are nearly identical (and not greater as expected) can be ascribed to the fact that when doing an analyze on the query we see, the tables are kept in main memory.

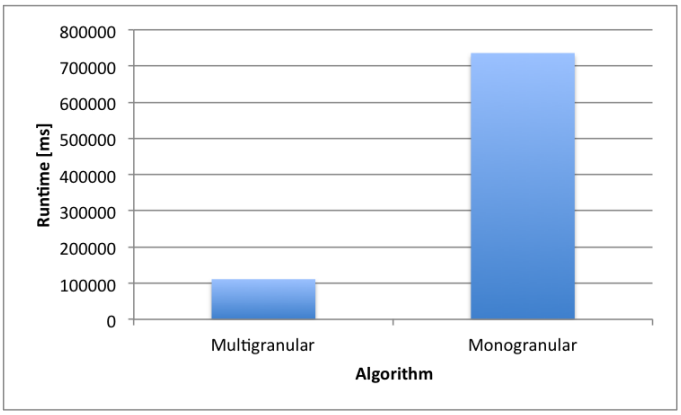


Figure 20: Runtime comparison unclustered distribution

6 Conclusion

In this report an extension of the nearest neighbor join operator has been implemented for dealing with similarity on timestamps. The TNNJ (temporal nearest neighbor join) operator computes an equijoin for two given relations $\mathbf{r}$ of schema $R = [E, G, T]$ and $\mathbf{s}$ of schema $S = [E, G, T, M]$ on E, G and a nearest neighbor join on T with group G for the tuples of the outer relation $\mathbf{r}$ without equijoin.

Since the tuples stored in $\mathbf{r}$ and $\mathbf{s}$ consist of multigranular timestamps we first gave a definition of the granularity, saying that for two tuples with timestamps T_1 and T_2 the tuple with the smaller granularity has the shortest intervall. Afterwards we introduced the Mulitgranular Euclidean Distance function for calculating the distance between multigranular timestamps, i.e. intervals.

By introducing partitions we achieved a total order of the tuples in $\mathbf{r}$ and $\mathbf{s}$ (inside one partition there exist a total order for the tuples) and a sort merge algorithm can then be run. We built the partitions by sorting the relations $\mathbf{r}$ and $\mathbf{s}$ by the granularity *gran*: the nearest neighbors for the tuples in $\mathbf{r}$ are always consecutive in a partition $\mathbf{s}_j \subseteq \mathbf{s}$.

The algorithm itself now joins every tuple in $\mathbf{r}$ with its *local* nearest neighbors, i.e. the nearest neighbors in every partition $\mathbf{s}_j$. At the end, once all local nearest neighbors have been found in every $\mathbf{s}_j$, we select as nearest neighbors only the tuples with the smallest distance.

In the experiment we show that computing a (multigranular) TNNJ using sort merge is faster than computing many (monogranular) joins between singular partitions. This is because our sort merge algorithm performs a sequential scan to fetch all tuples once at the beginning, while the monogranular algorithm has to jump from block to block multiple times to fetch the tuples of all partitions.